BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

```
#
#       scheme_table
#
#       specification for the compression scheme
#

# to specify a particular type,
# "type"  <type name>    <type bitwidth>

type BINARY_UNGUARDED_SHORT 26

# dst_offset     src_offset       width

0                SRC_1            7
7                SRC_2            7
14               DST                       7
21               OPCODE           3
24               OPCODE+3         2

endtype

#

type UNARY_PARAM7_UNGUARDED_SHORT 26

# dst_offset     src_offset       width

0                SRC_1            7
7                MODIFIER         7
14               DST                       7
21               OPCODE           3
24               OPCODE+3         2

endtype

#

type BINARY_PARAM7_RESULTLESS_UNGUARDED_SHORT 26

# dst_offset     src_offset       width

0                SRC_1            7
7                SRC_2            7
14               MODIFIER         7
21               OPCODE           3
24               OPCODE+3         2

endtype

#

type UNARY_SHORT 26

# dst_offset     src_offset       width

0                SRC_1            7
7                DST                       7
14               GUARD            7
21               OPCODE           3
24               OPCODE+3         2

endtype

#
```

```
type BINARY_SHORT 34
0                SRC_1            7
7        SRC_2            7
14       GUARD            7
21               OPCODE           3
24               OPCODE+3         2
26               DST                       7
33               CONST_0          1

endtype

#

type UNARY_PARAM7_SHORT 34
0                SRC_1            7
7        MODIFIER         7
14       GUARD            7
21               OPCODE           3
24               OPCODE+3         2
26               DST                       7
33               CONST_0          1

endtype

#

type BINARY_PARAM7_RESULTLESS_SHORT 34
0                SRC_1            7
7        SRC_2            7
14       GUARD            7
21               OPCODE           3
24               OPCODE+3         2
26               MODIFIER         7
33               CONST_0          1

endtype

#

type BINARY_UNGUARDED 34
0                SRC_1            7
7        SRC_2            7
14       DST                       7
21               OPCODE           3
24               OPCODE+3         2
26               OPCODE+5         3
29               LATENCY          2
31               CONST_0          1
32               CONST_1          2

endtype

#
type BINARY_RESULTLESS 34
0                SRC_1            7
7        SRC_2            7
14       GUARD                     7
21               OPCODE           3
24               OPCODE+3         2
26               OPCODE+5         3
29               CONST_1          2
31               CONST_0          2
33               CONST_1          1

endtype
```

```
#
type UNARY_PARAM7_UNGUARDED 34
0               SRC_1           7
7       MODIFIER    7
14      DST                     7
21              OPCODE          3
24              OPCODE+3        2
26              OPCODE+5        3
29              SIGN            1
30              LATENCY         1
31              CONST_1         3

endtype
#

type UNARY 34
0               SRC_1           7
7       DST             7
14      GUARD           7
21              OPCODE          3
24              OPCODE+3        2
26              OPCODE+5        3
29              LATENCY         2
31              CONST_1         1
32              CONST_0         1
33              CONST_1         1

endtype
#


type BINARY_PARAM7_RESULTLESS 42
0               SRC_1           7
7       SRC_2       7
14      GUARD           7
21              OPCODE          3
24              OPCODE+3        2
26              OPCODE+5        3
29              SIGN            1
30              CONST_1         3
33              CONST_0         2
35              MODIFIER        7

endtype
#

type BINARY 42
0               SRC_1           7
7       SRC_2       7
14      GUARD           7
21              OPCODE          3
24              OPCODE+3        2
26              OPCODE+5        3
29              LATENCY         2
31              CONST_0         1
32              CONST_1         1
33              CONST_0         1
34              CONST_1         1
35              DST                 7
endtype
#

type UNARY_PARAM7 42
0               SRC_1           7
7       MODIFIER    7
14      GUARD           7
21              OPCODE          3
24              OPCODE+3        2
26              OPCODE+5        3
29              SIGN            1
30              LATENCY         1
31              CONST_1         2
33              CONST_0         1
34              CONST_1         1
35              DST                 7
endtype
#

type ZEROARY_PARAM32 42
0               MODIFIER+7      7
7       MODIFIER    7
14      DST                     7
21              MODIFIER+14     3
24              MODIFIER+17     2
26              MODIFIER+19     5
31              CONST_1         3
34              MODIFIER+24     8
endtype
#

type ZEROARY_PARAM32_RESULTLESS 42
0               MODIFIER+7      7
7       MODIFIER    7
14      GUARD           7
21              MODIFIER+14     3
24              MODIFIER+17     2
26              MODIFIER+19     5
31              CONST_0         3
34              MODIFIER+24     8
endtype
#

type ADDRESS_1 32
0       7       7
7   0   7
14  21  3
17      24      2
19      26      5
24      34      8
endtype
#



# to alias a type,
# "alias"  <new typename>  <actual typename>

alias ZEROARY UNARY
alias ZEROARY_RESULTLESS UNARY
alias UNARY_RESULTLESS UNARY
alias UNARY_PARAM7_RESULTLESS UNARY_PARAM7

alias BINARY_RESULTLESS_SHORT BINARY_RESULTLESS
alias ZEROARY_PARAM32_SHORT ZEROARY_PARAM32
alias ZEROARY_PARAM32_RESULTLESS_SHORT ZEROARY_PARAM32_RESULTLESS
alias ZEROARY_SHORT UNARY
alias ZEROARY_RESULTLESS_SHORT UNARY
alias UNARY_RESULTLESS_SHORT UNARY
alias UNARY_PARAM7_RESULTLESS_SHORT UNARY_PARAM7_SHORT

alias BINARY_RESULTLESS_UNGUARDED BINARY_RESULTLESS
alias UNARY_UNGUARDED UNARY
```

```
alias BINARY_PARAM7_RESULTLESS_UNGUARDED BINARY_PARAM7_RESULTLESS
alias ZEROARY_PARAM32_UNGUARDED ZEROARY_PARAM32
alias ZEROARY_PARAM32_RESULTLESS_UNGUARDED ZEROARY_PARAM32_RESULTLESS
alias ZEROARY_UNGUARDED UNARY
alias ZEROARY_RESULTLESS_UNGUARDED UNARY
alias UNARY_RESULTLESS_UNGUARDED UNARY
alias UNARY_PARAM7_RESULTLESS_UNGUARDED UNARY_PARAM7_UNGUARDED
alias BINARY_RESULTLESS_UNGUARDED_SHORT BINARY_UNGUARDED_SHORT
alias UNARY_UNGUARDED_SHORT UNARY_SHORT
alias ZEROARY_PARAM32_UNGUARDED_SHORT ZEROARY_PARAM32
alias ZEROARY_PARAM32_RESULTLESS_UNGUARDED_SHORT ZEROARY_PARAM32_RESULTLESS
alias ZEROARY_UNGUARDED_SHORT UNARY
alias ZEROARY_RESULTLESS_UNGUARDED_SHORT UNARY
alias UNARY_RESULTLESS_UNGUARDED_SHORT UNARY
alias UNARY_PARAM7_RESULTLESS_UNGUARDED_SHORT UNARY_PARAM7_UNGUARDED_SHORT

alias UNARY_LONG BINARY
alias BINARY_LONG BINARY
alias BINARY_RESULTLESS_LONG BINARY
alias UNARY_PARAM7_LONG UNARY_PARAM7
alias UNARY_PARAM7_RESULTLESS_LONG UNARY_PARAM7
alias BINARY_PARAM7_RESULTLESS_LONG BINARY_PARAM7_RESULTLESS
alias ZEROARY_PARAM32_LONG ZEROARY_PARAM32
alias ZEROARY_PARAM32_RESULTLESS_LONG ZEROARY_PARAM32_RESULTLESS
alias ZEROARY_LONG BINARY
alias ZEROARY_RESULTLESS_LONG BINARY
alias UNARY_RESULTLESS_LONG BINARY

alias NOP_FORMAT BINARY

alias ZEROARY_PARAM32_ADDRESS  ADDRESS_1
alias ZEROARY_PARAM32_RESULTLESS_ADDRESS ADDRESS_1
alias ZEROARY_PARAM32_SHORT_ADDRESS  ADDRESS_1
alias ZEROARY_PARAM32_RESULTLESS_SHORT_ADDRESS ADDRESS_1
alias ZEROARY_PARAM32_UNGUARDED_ADDRESS  ADDRESS_1
alias ZEROARY_PARAM32_RESULTLESS_UNGUARDED_ADDRESS ADDRESS_1
alias ZEROARY_PARAM32_UNGUARDED_SHORT_ADDRESS  ADDRESS_1
alias ZEROARY_PARAM32_RESULTLESS_UNGUARDED_SHORT_ADDRESS ADDRESS_1
alias ZEROARY_PARAM32_LONG_ADDRESS  ADDRESS_1
alias ZEROARY_PARAM32_RESULTLESS_LONG_ADDRESS ADDRESS_1
#

#Address format are given below for each of the
#format types.
```

```
/*****************************************************************/
/*                      File:comp_shuffle.c
                         */
/*          Author: Hari Hampapuram                              */
/*****************************************************************/
/*
        Functions defined in this file:

                1. void shuffle_map_init(ScatterDescr *shuffle_map);
                2. void bits_shuffle(Module md, ReferenceTable *ref_table,
                                     CompressedBitstring *comp_bitstring)

*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/******* files from linker **********/
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"

/******* files from compressor *********/
#include "compressor.h"
#include "comp_reference.h"
#include "comp_scatter.h"
#include "comp_utils.h"

#define BLOCK_SIZE 256
void shuffle_map_init(ScatterDescr *shuffle_map)
{
int i;
int m_i,m_i_prime, row_num, byte_num_in_row, bit_position_in_byte;
int i_prime;
        shuffle_map->num_entries = BLOCK_SIZE;
        shuffle_map->total_width = BLOCK_SIZE;
        for (m_i_prime=0; m_i_prime < BLOCK_SIZE; m_i_prime++){
          /*
             Assuming the 256 bits of shuffled_word are numbered
             0 thru 255, we are trying to find the position of the
             ith bit from the left. But this ith bit will actually
             reside at a different position i' because the bits are
             reversed within a byte (within a byte, the lsb is to the
             right). i' = i/8 + (7 - i%8).
             Now i has to goto 'i) (say) according to the documnet.
             But, m(i) is actually located at m(i)' = m(i)/8 +(7- m(i)%8).
             In this descriptor, we need to provide a mapping from
             i' to m(i)'.
             The steps to take are:
           ' 1. Find out for each m(i)' (of the final text segment block)
             the m(i) of the document.
             2. Find out i as per the document, i.e find out where the
             bit m(i) comes from in the unshuffled text.
             3. Find out i', the position to which i goes to
             when the bits are reversed within a byte.
             4. make i' the src_offset, and m(i)' the dst_offset of m(i)'.
             */
           m_i = (m_i_prime / 8)*8 + (7 - m_i_prime % 8);
           row_num = m_i / 64;
           byte_num_in_row = (m_i % 64)/8;
           bit_position_in_byte = m_i % 8;
           i = (4*bit_position_in_byte + row_num)*8 +byte_num_in_row;
           i_prime = (i /8)*8 + (7 - i % 8);
#if 1
           shuffle_map->table[m_i_prime].dst_offset = m_i_prime;
           shuffle_map->table[m_i_prime].width = 1;
           shuffle_map->table[m_i_prime].src_offset = i_prime;
#else
           shuffle_map->table[m_i_prime].dst_offset = m_i_prime;
           shuffle_map->table[m_i_prime].width = 1;
           shuffle_map->table[m_i_prime].src_offset = 1 +  m_i_prime;
           if (m_i_prime == BLOCK_SIZE-1) shuffle_map->table[m_i_prime].src_offset = 0;
#endif
        }
}
/*
        step 1: For every BLOCK_SIZE bit chunk do the shuffling of the bits.
        This can be done by having a local scatter descriptor which is
        large enough, initializing it appropriately and then using
        extract and arrange bits.

        step 2: update the reference table of the text section
        to take care of the shuffling. This will also invlolve
        updating the scatter table.
*/
/* ---------------------------------------------------------------*/
/*
        for (word_num = 0; word_num < appropriate_size; word_num +=BLOCK_SIZE){
                extract_and_arrange_bits();
        }

        for (each entry in reference table){
                1. find out the word num i such that the position
                field of the reference is i*BLOCK_SIZE <= position <(i+1)*BLOCK_SIZE ;
                1a. set the position field to be i*BLOCK_SIZE;
                2. for (each position from 0 to 31){
                        1.      src_pos = get_position_in_src(of that position);
                        2.  if (the src_pos is in the range given in 1){
                                        set the new position by computing the
                                        position of (position - i*BLOCK_SIZE + src_pos)
in the
                                        shuffled word.
                                } else {
                                        set the new position by computing the
                                        position of (position - (i+1)*BLOCK_SIZE +src_po
s) and
                                        add BLOCK_SIZE to this to get the offset from i*
BLOCK_SIZE;
                                }
                        }
                3. Collapse the scatter descr found above, and
                insert into the  scatter table after making sure that
                it is not already there.
        }
*/
/* ---------------------------------------------------------------*/
```

```
void bits_shuffle(Module md, ReferenceTable *ref_table,
                  CompressedBitstring *comp_bitstring)
{


byte *addr;/* for iterating through teh compressed code. */
byte tmp_array[BLOCK_SIZE/8];/* for arranging bytes. */
int i;
RefDescr *current_reference;
unsigned long word_num;
ScatterDescr *scatter_ptr, *shuffle_map;
ScatterDescr *local_descr, *collapsed_descr;
int pos, src_pos, new_pos, position_offset;
int scat_id;


        local_descr = scatter_descr_create(MAX_SCATTER_TABLE_SIZE);
        collapsed_descr = scatter_descr_create(MAX_SCATTER_TABLE_SIZE);
        shuffle_map = scatter_descr_create(BLOCK_SIZE);
        shuffle_map_init(shuffle_map);
        for (addr = comp_bitstring->begin_address;
             addr < comp_bitstring->first_unused_address; addr += BLOCK_SIZE/8){
                extract_and_arrange_bits(addr, 0, shuffle_map, tmp_array);
                for (i=0; i < BLOCK_SIZE/8; i++){
                        *(addr + i) = tmp_array[i];
                }
        }

        reference_table_next(ref_table, TRUE);
        current_reference = reference_table_next(ref_table, FALSE);
        while (current_reference != NULL) {
                word_num = current_reference->position / BLOCK_SIZE;
                scatter_ptr =
                        scatter_descr_from_id(current_reference->scatter_type,
                                md->scatter_table);
                local_descr->num_entries = scatter_ptr->total_width;
                local_descr->total_width = scatter_ptr->total_width;
                position_offset =
                  current_reference->position - word_num*BLOCK_SIZE;
                for (pos=0; pos < scatter_ptr->total_width; pos++){
                        src_pos = get_position_in_src(pos, scatter_ptr);
                        if (src_pos + position_offset    < BLOCK_SIZE){
                                new_pos = src_pos_to_dst_pos(src_pos + position_offset,
shuffle_map);
                                /* Now new position is the offset of the given
                                bit w.r.t. the beginning of the block number
                                word_num.
                                */
                        }else{
                          new_pos =
                            src_pos_to_dst_pos(src_pos + position_offset - BLOCK_SIZE, s
huffle_map) + BLOCK_SIZE;
                        };

                        local_descr->table[pos].dst_offset = pos;
                        local_descr->table[pos].width = 1;
                        local_descr->table[pos].src_offset = new_pos;
                }
                current_reference->position = word_num * BLOCK_SIZE;
                collapse_scatter_descr(local_descr, collapsed_descr);
                scat_id = scatter_table_lookup(md->scatter_table,
                                               collapsed_descr);
                if (scat_id < 0) {
                        scatter_ptr = get_free_scatter_descr(md->scatter_table);
                        scatter_descr_cpy(scatter_ptr, collapsed_descr);
                        scat_id = scatter_table_lookup(md->scatter_table,
                                                       collapsed_descr);
                }
                current_reference -> scatter_type = scat_id;
                current_reference = reference_table_next(ref_table, FALSE);
        }
        scatter_descr_destroy(shuffle_map);
        scatter_descr_destroy(local_descr);
        scatter_descr_destroy(collapsed_descr);
}
```

```
/*****************************************************************/
/*          File: comp_scatter.c                                 */
/*          Author: Hari Hampapuram                              */
/*****************************************************************/
/* Associated files:
        This is does not use the functions in any other module.
*/

/* Functions defined in this module:
        ScatterTable *scatter_table_create(int);
        ScatterDescr * scatter_descr_from_id(int  , ScatterTable *);
        int scatter_table_lookup(ScatterTable *, ScatterDescr *);
        ScatterDescr *get_free_scatter_descr(ScatterTable *);
        scatter_descr_same(ScatterDescr *descr_1, ScatterDescr *descr_2);
        void scatter_descr_delete(ScatterTable *, ScatterDescr *);
        void scatter_descr_cpy(ScatterDescr *, ScatterDescr *);
        void  scatter_table_createAndLoad(Module  , ScatterTable **);
        byte * scatter_table_pack(byte *, ScatterTable *);
        int get_scatter_table_size(ScatterTable *);
        void scatter_table_cpy(ScatterTable *, ScatterTable *);
        void scatter_table_merge(ScatterTable *, ScatterTable *);
        int  get_position_in_src(int dst_pos, ScatterDesrc *scatter_ptr);
        void reverse_scatter_descr(ScatterDescr *src, ScatterDescr *dst)
        void collapse_scatter_descr(ScatterDescr *src, ScatterDescr *dst)
        void scatter_table_free(ScatterTable  *)
        ScatterDescr *scatter_descr_create(int table_size)
        ScatterDescr *scatter_descr_destroy(ScatterDescr *scatter_ptr)
        int src_pos_to_dst_pos(int src_pos, ScatterDescr *scatte_ptr)

*/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/******* files from linker **********/

#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "misc.h"
#include "scatter_types.h"

/*-----------------scatter_table_create()----------------------*/
/*
A table of scatter descriptors is created. Space for the
tables in these descriptors is not allocated here.
*/

ScatterTable *scatter_table_create(int size)
{
ScatterTable *scatter_ptr;
int i;


        if ((scatter_ptr = (ScatterTable *) malloc(sizeof(ScatterTable))) ==
                NULL){
                MALLOC_ERROR;
        }
        if ((scatter_ptr -> table = (Scatte Descr *)malloc(size*sizeof(ScatterDescr))) =
= NULL){
                MALLOC_ERROR;
        }
        scatter_ptr -> num_entries = 0;
        scatter_ptr -> capacity = size;

        for (i=0; i < size; i++){
                scatter_ptr -> table[i].num_entries = -1;
                scatter_ptr -> table[i].total_width = 0;
                scatter_ptr -> table[i].table = NULL;
        }
        return(scatter_ptr);
}

/*-------------------scatter_descr_from_id()-------------------------*/
/*
        Scatter table entries are to be accessed through scatter ids
        (also called scatter types). This function returns a pointer
        to a scatter descriptor given the scatter type.
*/
ScatterDescr *scatter_descr_from_id(int scat_id, ScatterTable *s_table)
{
        return(&(s_table -> table[scat_id]));
}

/*--------------------scatter_table_lookup()-------------------------*/
/*
        If s_table has an entry that is exactly the same as the one
        pointed to by s_item, returns the scatter type of such an
        entry. Otherwise returns -1.
*/

int scatter_table_lookup(ScatterTable *s_table,
                                ScatterDescr *s_item)
{
int i=0;

        if (s_item == NULL) return(-1);
        while (i < s_table -> capacity){
                if (scatter_descr_same(&(s_table -> table[i]), s_item)){
                        return(i);
                }
                i++;
        }
        return(-1);
}

/*--------------------get_free_scatter_descr()---------------------*/
/* find a free scatter descriptor in s_table. If none available,
realloc scatter table. Within the free scatter descriptor, malloc
space for a scatter table. return the sactter descriptor.
*/

ScatterDescr *get_free_scatter_descr(ScatterTable *s_table)
{
int i;
ScatterDescr *ptr;

        if (s_table -> capacity == s_table -> num_entries){
                ptr = (ScatterDescr *)realloc(s_table -> table, 2*(s_table -> capacity)*
sizeof(ScatterDescr));
                if (ptr == NULL){
                        MALLOC_ERROR;
                }
                s_table -> table = ptr;
                s_table -> capacity = 2*s_table -> capacity;
                for (i=s_table -> num_entries; i < s_table -> capacity; i++){
                        s_table -> table[i].num_entries = -1;
                        s_table -> table[i].total_width = 0;
                        s_table -> table[i].table = NULL;
                }
        }
```

```
        s_table ->  num_entries++;/* Another entry into the table assumed.*/
        for (i=0; i < s_table -> capacity; i++){
                if (s_table -> table[i].num_entries < 0){
                        break;
                }
        }
        s_table -> table[i].num_entries = 0;
        s_table -> table[i].total_width = 0;
        s_table -> table[i].table =
                (ScatterTriple *)malloc(sizeof(ScatterTriple)*MAX_SCATTER_TABLE_SIZE);
        if (s_table -> table[i].table == NULL){
                LOG_ERROR("NULL pointer encountered in get_free_scatter_descr().");
        }
        return(&(s_table -> table[i]));
}

/*-----------------------scatter_descr_same()----------------------*/
/*
return 1 if both descriptors are identical, i.e agree in the num
of entries, and the triples are identical.
 */

scatter_descr_same(ScatterDescr *descr_1, ScatterDescr *descr_2)
{

int i;

        if ((descr_1 == NULL)||(descr_2 ==  NULL)){
                return(0);
        }
        if (descr_1 -> num_entries != descr_2 -> num_entries){
                return (0);
        }
        for (i=0; i < descr_1 -> num_entries; i++){
                if (descr_1 -> table[i].src_offset != descr_2 -> table[i].src_offset)
                        return(0);
                if (descr_1 -> table[i].dst_offset != descr_2 -> table[i].dst_offset)
                        return(0);
                if (descr_1 -> table[i].width != descr_2 -> table[i].width)
                        return(0);
        }
        return(1);
}

/*-------------------------scatter_descr_delete()-----------------*/
/*
Delete the entry s_item in s_table. Free the space for table
in s_item and mark s_item with -1 (FREE_ENTRY)
in the num_entries column.
*/

void
scatter_descr_delete(ScatterTable *s_table, ScatterDescr *s_item)
{
int i;

        for (i=0; i < s_table -> num_entries; i++){
                if (&(s_table -> table[i]) == s_item){
                        break;
                }
        }
        if (&(s_table -> table[i]) != s_item){
                LOG_ERROR("Trying to delete nonexistant scatter table entry.");
        }
        free(s_item -> table);
        s_item -> num_entries = -1;
        s_item -> table = NULL;
        s_item -> total_width = 0;
        s_table -> num_entries--;
}

/*-----------------------scatter_descr_cpy()----------------------*/
/*
        Copy the values of all the fields from item_src to item_dst.
        It is assumed that item_dst has sufficient space to hold
        the values from item_src.
*/

void
scatter_descr_cpy(ScatterDescr *item_dst, ScatterDescr *item_src)
{
int i;

        item_dst -> num_entries = item_src -> num_entries;
        item_dst -> total_width = item_src -> total_width;
        for (i=0; i < item_src -> num_entries; i++){
                item_dst -> table[i].src_offset = item_src -> table[i].src_offset;
                item_dst -> table[i].dst_offset = item_src -> table[i].dst_offset;
                item_dst -> table[i].width = item_src -> table[i].width;
        }

}

#if 0
/*-----------------------scatter_table_createAndLoad()----------------------*/
/*
Reads the scatter table from the memory image of md into
s_table. Space for s_table is allocated here. The only precondition
is that the module structure md must have been initialized to
have the values of md -> global_image, md -> scatter_tbl_offs and
the memory image of the global partition must be available.
*/

void
scatter_table_createAndLoad(Module md, ScatterTable **s_table)
{

        byte *mptr = md -> global_image + md -> scatter_tbl_offs;
        unsigned long num_entries, i, j;
        ScatterDescr *s_descr;

        num_entries = Size4_get(mptr);
        mptr += 4;
        *s_table = scatter_table_create(num_entries);
        for (i=0; i < num_entries; i++){
                s_descr = get_free_scatter_descr(*s_table);
                s_descr -> num_entries  = Size4_get(mptr);
                mptr += 4;
                if (s_descr -> num_entries > MAX_SCATTER_TABLE_SIZE) {
                        LOG_ERROR("Number of entries in scatter descriptor too large.");

                }
                s_descr -> total_width = 0;
                for (j=0; j < s_descr -> num_entries; j++){
                        s_descr -> table[j].dst_offset =
                                Size4_get(mptr + LIFE_Obj_scatter_dst_offset);
                        s_descr -> table[j].width =
                                Size4_get(mptr + LIFE_Obj_scatter_width);
                        s_descr -> total_width += s_descr -> table[j].width;
                        s_descr -> table[j].src_offset =
                                Size4_get(mptr + LIFE_Obj_scatter_src_offset);
                        mptr += 12;
```

```
                )
        }
}
#endif


/*-------------------scatter_table_pack()-----------------------*/
/*
        Pack the s_table so into memory starting at mptr, so that
        it is ready to be output as part of the final object module.
        This returns the pointer to theh location from where subsequent
        portions of the object module can be written.
*/

byte *
scatter_table_pack(byte *mptr, ScatterTable *s_table)
{
int i, j;
ScatterDescr *s_descr;

        Size4_put(mptr, s_table -> num_entries);
        mptr +=4;
        for (i=0; i < s_table -> num_entries; i++){
                s_descr = &(s_table -> table[i]);
                Size4_put(mptr, s_descr -> num_entries);
                mptr +=4;
                for (j=0; j < s_descr -> num_entries; j++){
                        Size4_put(mptr + LIFE_Obj_scatter_dst_offset, s_descr -> table[j
].dst_offset);
                        Size4_put(mptr + LIFE_Obj_scatter_width, s_descr -> table[j].wid
th);
                        Size4_put(mptr + LIFE_Obj_scatter_src_offset, s_descr -> table[j
].src_offset);
                        mptr +=12;
                }
        }
        return(mptr);
}

/*-------------------get_scatter_table_size()-----------------------*/
/*
        returns the number of bytes required for the scatter table
        in the object module.
*/

int
get_scatter_table_size(ScatterTable *s_table)
{
int size, i;
        size = 4;/* for number of descriptors in the table. */
        for (i=0; i < s_table -> num_entries; i++){
                size += 4; /* for num triples in the descriptor. */
                size += s_table -> table[i].num_entries * 12;
                        /* 12 bytes per triple. */
        }
        return(size);
}

/*---------------scatter_table_cpy()----------------------*/
/*
        Copies the scatter_table entries from src to dst. It is ssumed
        that dst is currently initialized but has no entries in it.
        This routine is required in the compressor for copying the
        scatter table from the input module to the output module.
        By copying the scatter table from input to output right at the
        beginning, the scatter types of all the bitfields that are not
        affected by the compressor remain the same.
*/

void
scatter_table_cpy(ScatterTable *dst, ScatterTable *src)
{
ScatterDescr *scatter_ptr;
unsigned long i;
        for (i=0; i < src -> num_entries; i++){
                scatter_ptr = get_free_scatter_descr(dst);
                scatter_descr_cpy(scatter_ptr, &(src -> table[i]));
        };
}

/*-------------------scatter_table_merge()-----------------------*/
/*
        This is required for the linker. The entries from the from_table
        are entered into the into_table.
        NOTE: As a result of the merging, the entries in the reference
        tables of the various sections in the module having the from
        table will change. Thus they should be updated.
*/

void
scatter_table_merge(ScatterTable *into_table, ScatterTable *from_table)
{

int i;
ScatterDescr *scatter_ptr, *new_ptr;

        for (i=0; i < from_table -> num_entries; i++){
                scatter_ptr = &(from_table -> table[i]);
                if (scatter_table_lookup(into_table, scatter_ptr) < 0){
                        new_ptr = get_free_scatter_descr(into_table);
                        scatter_descr_cpy(new_ptr, scatter_ptr);
                }
        }
}


/*-------------------get_position_in_src()-----------------------*/
/*
        Given a position i in the destination bit vector dv of a scatter
        descriptor, get the position in the source bit vector from
        where dv[i] is to be fetched.
*/

int
get_position_in_src(int dst_pos, ScatterDescr *scatte_ptr)
{
int i;
ScatterTriple *triple;

        for(i=0; i<scatte_ptr->num_entries; i++){
                triple = &(scatte_ptr->table[i]);
                if ((dst_pos >= triple->dst_offset) &&
                                (dst_pos < triple->dst_offset + triple->width)){
                        return(triple->src_offset + dst_pos - triple->dst_offset);
                }
        }
        return(-1);
}

/*---------------------------src_pos_to_dst_pos()-------------*/
/*
   Given a position in the src vector, this returns the position
```

```
to which the the bit is moved to in the destination vector.
*/
int
src_pos_to_dst_pos(int src_pos, ScatterDescr *scatte_ptr)
{
int i;
ScatterTriple *triple;

        for(i=0; i<scatte_ptr->num_entries; i++){
                triple = &(scatte_ptr->table[i]);
                if ((src_pos >= triple->src_offset) &&
                                (src_pos < triple->src_offset + triple->width)){
                        return(triple->dst_offset + src_pos - triple->src_offset);
                }
        }
        return(-1);
}

/*-----------------reverse_scatter_descr()---------------------*/
/*
        Given a scatter descriptor, that specifies how to get the
        bits of a vector v[0...N], this function gives the scatter
        descriptor for getting the vector w[0...N] = v[N...0] from
        the same initial source vector.
*/

void
reverse_scatter_descr(ScatterDescr *src, ScatterDescr *dst)
{
int i, dst_pos, src_pos;

        for (i=src->total_width-1; i >=0; i--) {
                dst_pos = src->total_width-1 -i;
                        /* get the position in source
                                vector of the ith bit of v. */
                if ((src_pos = get_position_in_src(i,src)) < 0){
                        LOG_ERROR("Illegal position returned by get_position_in_src().")
;
                }
                dst->table[dst_pos].dst_offset = dst_pos;
                dst->table[dst_pos].width = 1;
                dst->table[dst_pos].src_offset = src_pos;
        }
        dst->num_entries = src->total_width;
        dst->total_width = src->total_width;
}

void
collapse_scatter_descr(ScatterDescr *src, ScatterDescr *dst)
{
int i, dst_pos, src_pos;
int current_dst_idx, current_width, prev_pos;
/* first of all put the src into a "split form", i.e.,
compute into dst a scatter descr that is equivalent to src
but specifies each dst position separately (all widths = 1).
        Next go through the split form and combine fields together
whenever possible.
*/
        for (dst_pos=0; dst_pos < src->total_width;  dst_pos++) {
                if ((src_pos = get_position_in_src(dst_pos,src)) < 0){
                        LOG_ERROR("Illegal position returned by get_position_in_src().")
;
                }
                dst->table[dst_pos].dst_offset = dst_pos;
                dst->table[dst_pos].width = 1;
                dst->table[dst_pos].src_offset = src_pos;
        }

        dst->total_width = src->total_width;
        current_dst_idx = 0;
        current_width = 1;
        prev_pos = dst->table[0].src_offset;
        for (dst_pos=1; dst_pos < src->total_width;  dst_pos++) {
                if (dst->table[dst_pos].src_offset == prev_pos+1){
                        current_width++;
                        prev_pos++;
                }else{
                        dst->table[current_dst_idx++].width = current_width;
                        dst->table[current_dst_idx].dst_offset = dst_pos;
                        dst->table[current_dst_idx].src_offset =
                                dst->table[dst_pos].src_offset;
                        prev_pos = dst->table[dst_pos].src_offset;
                        current_width = 1;
                }
        }
        /* The last current_dst_idx was not handled in the loop.
                Do it here. */
        dst->table[current_dst_idx++].width = current_width;

        dst->total_width = src->total_width;
        dst->num_entries = current_dst_idx;
}

void
scatter_table_free(ScatterTable *s_table)
{
int i;

        for (i=0; i < s_table -> capacity; i++){
                if (s_table -> table[i].num_entries != -1) {
                        free(s_table -> table[i].table);
                }
        }
        /* All the memory allocated within the scatter descriptors
                to store the scatter triples is now freed. We can go
                ahead and free teh space taken up by the scatter table.
        */
        free(s_table -> table); /* free the array of scatter descriptors*/
        free(s_table);
}

ScatterDescr *scatter_descr_create(int table_size)
{
ScatterDescr *scatter_ptr;

        if ((scatter_ptr = (ScatterDescr *)malloc(sizeof(ScatterDescr))) == NULL){
                MALLOC_ERROR;
        };
        scatter_ptr->num_entries = 0;
        scatter_ptr->total_width = 0;
        if ((scatter_ptr->table = (ScatterTriple *)malloc(table_size*sizeof(ScatterTripl
e)))
                == NULL){
                MALLOC_ERROR;
        }
        return(scatter_ptr);
}

void scatter_descr_destroy(ScatterDescr *scatter_ptr)
{
        free(scatter_ptr->table);
        free(scatter_ptr);
```

```
/*************************************************************/
/*          File: comp_scheme.c                              */
/*          Author: Hari Hampapuram                          */
/*************************************************************/
/*
        Associated files:
                1. scheme.c - for initializing the format table.
                2. tables.c - for initializing opcode information
                3. comp_utils.c
                4. comp_scatter.c
*/

/* Functions defined in this file:
        1. static local_extract_and_arrange_bits(byte *src_base,
                                int offset_in_byte, ScatterDescr *scatter_ptr,
                                byte *dst_base, int sign_bit, int latency_bit)
        2. static void get_ith_operation(byte *inst,
                                OperationDescr * oper, unsigned int issueslot)
        3. static fmtid_to_scatter_index(int format_id,
                                FormatTable *format_table)
        4. static scatter_id_to_fmtid(int scatter_id,
                                FormatTable *format_table)
        5. static get_address_format(int scatter_id,
                                FormatTable *format_table)
        6. static get_size(int format_id, FormatTable *format_table)
        7. static get_ith_size(CompressedIns *comp_ins, int i)
        8. static compute_format_id(OperationDescr *operation,
                FormatTable *format_table, boolean is_btarget)
        9. static void compute_total_widths(FormatTable *fmt_table)
        10. static void get_context_bits(CompressedIns *comp_ins,
                                FormatTable *format_table
        11. void get_compressed_ops(byte *input_ins,
                        CompressedIns *compressed_ins, boolean is_btarget)
        12. byte * get_address_for_next(CompressedBitstring *comp_bitstring,
                        boolean is_btarget)
        13. void dressup_ins(CompressedIns *comp_ins, byte *context_bits,
                ScatterTable *input_s_table)
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/******* files from linker **********/
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"

/******* files from compressor *********/
#include "compressor.h"

#include "comp_utils.h"
#include "comp_scatter.h"
#include "comp_scheme.h"
#include "opcode.h"


int ON_THE_FLY = 0;

static FormatTable FORMAT_TABLE;
static LongCodes LONG_CODES[NUM_OPCODES];
static unsigned char SHORT_CODES[NUM_OPCODES];
static OP_TYPE[NUM_OPCODES];
static IS_SHORT[NUM_OPCODES];
static opcode_sign[NUM_OPCODES];
static opcode_latency[NUM_OPCODES];
static unsigned char address_opcode[NUM_OPCODES];
static byte CONST_FORMAT[] = {0x55,0x55};
static byte CBITS_00 = 0x00;
static byte CBITS_01 = 0x01;
static byte CBITS_10 = 0x02;
static byte CBITS_11 = 0x03;
/*-----------local_extract_and_arrange_bits()-----------*/
/*
        This is very similar to the extract_and_arrange function in
        comp_utils.c. This is here because it handles the SIGN and
        LATENCY bits also, which is part of the compression scheme.
*/

static
local_extract_and_arrange_bits(byte *src_base, int offset_in_byte,
          ScatterDescr *scatter_ptr, byte *dst_base,
                        int sign_bit, int latency_bit)
{
/* The source is specified by giving a byte *src_base and an
offset within the byte. The scatteer descriptor is to be
interpreted starting from the position offset_in_byte within
src_base.  This is handled this way because (for example) the
assembler gives teh scatter descr. w.r.t. some position in the
bitstring and this need not be aligned with a byte. We compute
the byte within which the position falls and pass that byte * with
the appropriate offset. We should be having a dst_offset too
for consistency!*/
int i;
ScatterTriple triple;
int sign_val, latency_val;

        if (sign_bit == 0) sign_val = CONST_0; else sign_val = CONST_1;
        if (latency_bit == 0) latency_val = CONST_0; else latency_val = CONST_1;
    for (i=0; i < scatter_ptr -> num_entries; i++){
        triple = scatter_ptr -> table[i];
                /*replace below with a switch.*/
        if (triple.src_offset == CONST_0){
            fill_const(dst_base, triple.dst_offset, triple.width,CONST_0);
        } else {
            if (triple.src_offset == CONST_1) {
                fill_const(dst_base, triple.dst_offset, triple.width,CONST_1);
            }else{
                                if (triple.src_offset == SIGN){
                                        fill_const(dst_base, triple.dst_offset,triple.wi
dth,sign_val);
                                }else{
                                if (triple.src_offset == LATENCY){
                                        fill_const(dst_base, triple.dst_offset,triple.wi
dth,latency_val);
                                }else{
                bits_translate(src_base, triple.src_offset + offset_in_byte,
```

```
                    triple.width, dst_base, triple.dst_offset);
                                )
            )
           )
    )
        )
)

/*-------------------get_ith_operation()-------------------------*/
/*
        inst - must be a byte ptr to a location from where an uncompressed
                        instruction starts. (By uncompressed, I mean one in
                        canonical form, as produced by the assembler.)
                        (The modification below - the ON_THE_FLY part is a quick
                        and dirty modification to facilitate cpu testing and so on
                        where the order of the various fields is different.)
        oper - must be a valid ptr to a OperationDescr struct.
        issueslot must be from 0..NUM_ISSUE_SLOTS-1.

        The function populates the struct *oper with the values from
        the issueslot number issueslot of inst.
*/

void get_ith_operation(byte *inst, OperationDescr * oper, unsigned int issueslot)
{
int i;
     unsigned int pos;
/*,
         printf("get_ith_operation(issueslot=%d)\n", issueslot);
         for (i=0; i<INSTRUCTION_WIDTH_BYTES; i++){
                write_byte(stdout, inst[i]);
                printf(" ");
         }
         printf("\n");
*/
     pos = issueslot * OPERATION_WIDTH;
         oper->opcode = 0;
         oper->guard = 0;
         oper->arg1 = 0;
         oper->arg2 = 0;
         oper->dest = 0;
        if(!ON_THE_FLY){
     bits_translate(inst, pos, OPCODE_WIDTH, (&(oper->opcode)), 0);
     pos += OPCODE_WIDTH;
     GET_BITS(inst, pos, MODIFIER_WIDTH, (oper->modifier), 0);
     pos += MODIFIER_WIDTH;
     GET_BITS(inst, pos, GUARD_WIDTH, (&(oper->guard)), 0);
     pos += GUARD_WIDTH;
     GET_BITS(inst, pos, ARG1_WIDTH, (&(oper->arg1)), 0);
     pos += ARG1_WIDTH;
     GET_BITS(inst, pos, ARG2_WIDTH, (&(oper->arg2)), 0);
     pos += ARG2_WIDTH;
     GET_BITS(inst, pos, DEST_WIDTH, (&(oper->dest)), 0);
        }else{
                bits_translate(inst, pos, OPCODE_WIDTH, (&(oper->opcode)), 0);
                pos += OPCODE_WIDTH;
                GET_BITS(inst, pos, ARG1_WIDTH, (&(oper->arg1)), 0);
                pos += ARG1_WIDTH;
                GET_BITS(inst, pos, ARG2_WIDTH, (&(oper->arg2)), 0);
                pos += ARG2_WIDTH;
                GET_BITS(inst, pos, GUARD_WIDTH, (&(oper->guard)), 0);
                pos += GUARD_WIDTH;
                GET_BITS(inst, pos, DEST_WIDTH, (&(oper->dest)), 0);
                pos += DEST_WIDTH;
                GET_BITS(inst, pos, MODIFIER_WIDTH, (oper->modifier), 0);
        }
}

/*-------------------fmtid_to_scatter_index()-------------------------*/
/*
        Given the format_id that is computed for a given operation
        i.e., as a function of the opcode, branch target information,
        guard register information, and the format table that is storing
        the compression formats, this function returns the index into
        the scatter table where the scatter descriptor for the
        format_id is located.
*/

static
fmtid_to_scatter_index(int format_id, FormatTable *format_table)
{
int i;

        for (i=0; i < format_table -> num_of_formats; i++){
                if (format_table -> id_to_idx_table[i].format_id == format_id){
                        return(format_table -> id_to_idx_table[i].scatter_table_index);
                }
        }
        return(-1);
}

/*--------------------scatter_id_to_fmtid()---------------------*/
/*
        Inverse of the above function.
*/

static
scatter_id_to_fmtid(int scatter_id, FormatTable *format_table)
{
int i;

        for (i=0; i < format_table -> num_of_formats; i++){
                if (format_table -> id_to_idx_table[i].scatter_table_index == scatter_id
){
                        return(format_table -> id_to_idx_table[i].format_id);
                }
        }
        return(-1);
}

/*-------------------get_address_format()----------------------------*/
/*
        Given the scatter_id (which is actually the id of a format
        outside the format table object), this function returns the
        scatter type of the address format if the format corresponding
        to scatter_id has an address field in it.
*/

static get_address_format(int scatter_id, FormatTable *format_table)
{
int address_fmt, idx, format;

        format = scatter_id_to_fmtid(scatter_id, format_table);
        address_fmt = format + ADDRESS_SHIFT;
        idx = fmtid_to_scatter_index(address_fmt, format_table);
        if (idx < 0) return(NO_ADDRESS);
        return(idx);
}

/*-------------------------get_size()----------------------------*/
/*
        returns the number of bits required for the format.
```

```
*/

static
get_size(int format_id, FormatTable *format_table)
{
        return(format_table -> size_table[format_id]);
}

/*--------------------get_ith_size()-------------------------*/
/*
        Given a compressed instruction, returns the number of bits
        required for the ith operation.
*/

static
get_ith_size(CompressedIns *comp_ins, int i)
{
int format_id;

        format_id = comp_ins -> operation[i].format;
        return(FORMAT_TABLE.size_table[format_id]);
}

/*--------------------compute_format_id()---------------------*/
/*
        Given an operation description and the branch target info, this
        computes the "internal" format id and the the corresponding
        scatter_id (which is the external foramt_id) and returns it.
*/

static
compute_format_id(OperationDescr *operation,
                FormatTable *format_table, boolean is_btarget)
{
int format;
int scatter_index;
byte tmp_guard=0;

        if (IS_NOP(operation -> opcode)){
                format = NOP_FORMAT;
        }else{
                if (!is_btarget){
                        GET_BITS(&(operation -> guard), 0,GUARD_WIDTH,
                                                &tmp_guard, 1);
                        format = OP_TYPE[operation -> opcode] +
                        IS_SHORT[operation -> opcode]*SHORT_SHIFT  +
                                IS_UNGUARDED(tmp_guard)*UNGUARDED_SHIFT;
                }else{
                        format = OP_TYPE[operation -> opcode] + LONG_SHIFT;
                }
        }

        scatter_index = fmtid_to_scatter_index(format, format_table);
        if (scatter_index == -1) {
                LOG_ERROR("Illegal scatter index retruned by fmtid_to_scatter_index().")
;
        }
        return(scatter_index);
}

/*--------------------compute_total_widths()-----------------*/
/*
        This function is used to initialize the total_widths fields
        of the format table.
*/

static void
compute_total_widths(FormatTable *fmt_table)
{
int i, j;

        for (i=0; i< fmt_table->num_of_formats; i++){
                fmt_table->scatter_table[i].total_width = 0;
                for(j=0; j < fmt_table->scatter_table[i].num_entries; j++){
                        fmt_table->scatter_table[i].total_width +=
                                fmt_table->scatter_table[i].table[j].width;
                }
        }

}

/*----------------------get_context_bits()-------------------------*/
/*
        Computes the foramt bits for the compressed information and
        initializes comp_ins->context_bits appropriately.
*/

static void
get_context_bits(CompressedIns *comp_ins, FormatTable *format_table)
{
int i;
OperationDescr * operation;
int num_bytes_in_context, size;

        num_bytes_in_context = (2*NUM_ISSUE_SLOTS)/8;
        if ((2*NUM_ISSUE_SLOTS)%8 != 0){
                num_bytes_in_context++;
        }
        if (comp_ins -> is_btarget) {
                for (i=0; i < num_bytes_in_context; i++){
                        comp_ins -> context_bits[i] = CONST_FORMAT[i];
                }
                return;
        }

        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                operation = &(comp_ins -> operation[i]);
                size = get_size(operation -> format, format_table);
                if (IS_NOP(operation -> opcode)){
                        ONE_ONE(i, comp_ins -> context_bits);
                } else {
                        if (size == 26){
                                ZERO_ZERO(i, comp_ins -> context_bits);
                        }
                        if (size == 34){
                                ONE_ZERO(i, comp_ins -> context_bits);
                        }
                        if (size == 42){
                                ZERO_ONE(i, comp_ins -> context_bits);
                        }
                }

        }
}

/*-------------------------get_compressed_ops()--------------------*/
/*
        input_ins - must point to an instruction in canonical form.
                                (or the form output by the memory dump of ls+)
        compressed_ins - this will get populated with the compressed
                                format of each operation. context_bits are also
                                computed here.
```

```
        is_btarget - is needed for computing the appropriate format.
*/

void
get_compressed_ops(byte *input_ins,
        CompressedIns *compressed_ins, boolean is_btarget)
{
unsigned int i;
OperationDescr *operation;
static first_time = 1;

        if (first_time) {
                format_table_init(&FORMAT_TABLE);
                compute_total_widths(&FORMAT_TABLE);
                optypes_init(LONG_CODES, SHORT_CODES, OP_TYPE,
                                        IS_SHORT,address_opcode,
                                        opcode_sign, opcode_latency);
                first_time = 0;
        }

        compressed_ins -> num_operations_used = 0;
        compressed_ins -> is_long_format = is_btarget;
        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                operation = &(compressed_ins -> operation[i]);
                get_ith_operation(input_ins, operation, i);
                operation -> comp_opcode[0] =0;
                if (is_btarget){
                        GET_BITS(&(LONG_CODES[operation -> opcode].lo), 0, 8,
                                &(operation -> comp_opcode[0]), 0);
                        operation -> is_short = FALSE;
                } else {
                        if (IS_SHORT[operation -> opcode]){
                                GET_BITS(&(SHORT_CODES[operation -> opcode]), 3, 5,
                                        &operation -> comp_opcode[0], 0);
                                operation -> is_short = TRUE;
                        } else {
                        GET_BITS(&(LONG_CODES[operation -> opcode].lo), 0, 8,
                                &(operation -> comp_opcode[0]), 0);
                                operation -> is_short = FALSE;
                        }
                }
                operation -> format =
                        compute_format_id(operation, &FORMAT_TABLE, is_btarget);

                if (!(IS_NOP(operation -> opcode))){
                        compressed_ins -> num_operations_used++;
                }
        }

        get_context_bits(compressed_ins, &FORMAT_TABLE);

}

/*--------------------get_address_for_next()----------------------*/
/*
        This computes where to start appending the next instruction
        in the compressed bitstring.
*/

byte *
get_address_for_next(CompressedBitstring *comp_bitstring,
        boolean is_btarget)
{
int pad_bytes=0;
int i;

        if (!G_all_globals.flags.padoff){
                if (!is_btarget){
                        return(comp_bitstring -> first_unused_address);
                }

                pad_bytes = (comp_bitstring -> first_unused_address -
                                        comp_bitstring -> begin_address) % BTARGET_ALIGN
_BYTES;
                if (pad_bytes != 0){
                        pad_bytes = BTARGET_ALIGN_BYTES - pad_bytes;
                }
                if (pad_bytes > LENGTH_OF_BTARGET_IN_BYTES){
                        pad_bytes = 0;
                }
                for (i=0; i < pad_bytes; i++){
                        *(comp_bitstring -> first_unused_address + i) = 0;
                }
        /* if pad_bytes is now large enough to hold the entire
        instruction, then just insert the instruction at first
        unused address by setting pad_bytes to 0. */
        }
        return(comp_bitstring -> first_unused_address + pad_bytes);
}

/*-----------------------dressup_ins()-----------------------------*/
/*
        Given all teh components of the final compressed instruction,
        as obtained by get_compressed_ops(), we still need to combine
        all the information, and put them into comp_ins->final_instruction
        in the way the instruction gets written into the compressed file.
        This involves combining the format bits and the operation bits,
        and moving bits around in the instruction based on the 24-bit
        part, 2-bit part and the extension part information.
        This also means that the scatter types of the addresses need
        recomputing and appropriate entries made in the output scatter
        table. (which is unfortunately called input_s_table here!)

        NOTE: Here the context_bits are from the next instruction.
        If context_bits is NULL CONST_FORMAT is used.
*/

void
dressup_ins(CompressedIns *comp_ins, byte *context_bits,
                        ScatterTable *input_s_table)
{
char error_str[MAX_LENGTH_OF_ERROR];

byte tmp_1[MAX_BYTES_IN_COMP_INS];
byte tmp_2[MAX_BYTES_IN_COMP_INS];
int i, j;
OperationDescr *op_ptr;
int size;/*tmp variable for size of an operation. */
ScatterDescr local_descr, *scat_ptr, new_descr,
                        reverse_descr, collapsed_descr;
int current_position; /*used for copying operrations. */
int scat_id; /*scatter id - used tc do scatter table lookup. */
ScatterTriple s_arr[MAX_SCATTER_TABLE_SIZE],
                                s_arr2[MAX_SCATTER_TABLE_SIZE],
                                s_arr3[MAX_SCATTER_TABLE_SIZE],
                                s_arr4[MAX_SCATTER_TABLE_SIZE];

byte nop_bits[] = {0xff};
int num_issue_slots_used,
        num_possible_ops_in_first,
        num_ops_in_first_group,
        padding_bits_first,
        num_ops_in_end,
```

```
        padding_bits_last,
        position,
        current_2bit_position,
        extension_position,
        current_opnum,
        total_num_bits_in_ins,
        src_pos,
        new_offset,
        pos,
        slot;
int extension_start[NUM_ISSUE_SLOTS],
        twenty_four_bit_start[NUM_ISSUE_SLOTS],
        two_bit_start[NUM_ISSUE_SLOTS];

        local_descr.table = s_arr;
        new_descr.table = s_arr2;
        reverse_descr table = s_arr3;
        collapsed_descr.table = s_arr4;
        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                op_ptr = &(comp_ins -> operation[i]);

                        tmp_1[SRC_1_BYTE] = op_ptr -> arg1;
                        tmp_1[SRC_2_BYTE] = op_ptr -> arg2;
                        tmp_1[DST_BYTE]   = op_ptr -> dest;
                        tmp_1[GUARD_BYTE] = op_ptr -> guard;
                        for (j=0; j < BYTES_PER_MODIFIER; j++){
                                tmp_1[MODIFIER_BYTE + j] = op_ptr -> modifier[j];
                        };
                        for (j=0; j < BYTES_PER_OPCODE; j++){
                                tmp_1[OPCODE_BYTE + j] = op_ptr -> comp_opcode[j];
                        };
                        local_extract_and_arrange_bits(tmp_1, 0,
                                &(FORMAT_TABLE.scatter_table[op_ptr -> format]),
                                tmp_2 + i*BYTES_IN_ONE_COMP_OP,
                                opcode_sign[op_ptr -> opcode],
                                opcode_latency[op_ptr -> opcode]);
/*
                        extract_and_arrange_bits(tmp_1, 0,
                                &(FORMAT_TABLE.scatter_table[op_ptr -> format]),
                                tmp_2 + i*BYTES_IN_ONE_COMP_OP);
*/
                        op_ptr -> address_field_format = get_address_format(op_ptr -> fo
rmat, &FORMAT_TABLE);
                        if (address_opcode[op_ptr -> opcode]) {
                                if (op_ptr -> address_field_format == NO_ADDRESS){
                                        sprintf(error_str,"Address field format not avai
lable for opcode %d.", op_ptr -> opcode);
                                        LOG_ERROR(error_str);
                                }
                        }
                        /* The above stmt gives the basic address format, within
                                an operation. i.e., the format assuming that the
                                bits in the operation are not reordered according to
                                twenty_four-bit part, 2-bit part etc. The reordering res
ults
                                in another level of scattering and that will be
                                handled by the reordering code.  The final address
                                format will be w.r.t. the beginning of the instruction.
                                */
                        if(op_ptr -> opcode == IJMPI_OPCODE){
                                fill_const(tmp_2 +i*BYTES_IN_ONE_COMP_OP,31 , 1,CONST_1)
;
                        }

        }
```

```
/* The BNF stuff goes here. This also handles the format bits.
        A NULL context_bits => CONST_FORMAT. */
/* for now we assume that there is no reordering. The context
        bits are placed first, and then the operations are
        placed one after the other. This would mean that the
        address formats now just get translated by some amount. */
        if (context_bits != NULL){
                bits_translate(context_bits, 0, 2*NUM_ISSUE_SLOTS,
                        comp_ins -> final_instruction, 0);
        } else {
                bits_translate(CONST_FORMAT, 0, 2*NUM_ISSUE_SLOTS,
                        comp_ins -> final_instruction, 0);
        };
        /* Determine the start positions of the 24-bit parts,
        2-bit parts, and the extensions for each of the issue slots.
        */
        num_issue_slots_used = 0;
        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                op_ptr = &(comp_ins -> operation[i]);
                if ((!(IS_NOP(op_ptr -> opcode))) ||
                        (comp_ins -> is_long_format == TRUE)){
                        num_issue_slots_used++;
                }
        }
        num_possible_ops_in_first = 4 - (NUM_ISSUE_SLOTS % 4);
        num_ops_in_first_group =
                (num_issue_slots_used <= num_possible_ops_in_first) ?
                        num_issue_slots_used:num_possible_ops_in_first;
        padding_bits_first =
                (num_issue_slots_used <= num_possible_ops_in_first) ?
                2*(num_possible_ops_in_first - num_ops_in_first_group):0;
        num_ops_in_end =
                (num_issue_slots_used - num_ops_in_first_group) % 4;
        padding_bits_last =
                ( num_ops_in_end > 0) ? 2*(4 - num_ops_in_end) : 0;
        position = 2*NUM_ISSUE_SLOTS + num_possible_ops_in_first*2;
        current_2bit_position = position - 2;
        extension_position = 2*NUM_ISSUE_SLOTS +
                num_issue_slots_used*26+ padding_bits_first +
                padding_bits_last;
        current_opnum = 0;
        /* position keeps track of the current position in the final
                layout. We find out where teh 24bit, 2bit, and extensions
                of each issue slot starts. current_2bit_position keeps
                track of where teh 2bit part of the next useful operation
                goes. extension_position keeps track of the current
                position for the extension.
        */
        total_num_bits_in_ins = extension_position;
        /* we will add the extension bit sizes to total_num_bits_in_ins
                in the loop below. */
        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                op_ptr = &(comp_ins -> operation[i]);
                if ((!(IS_NOP(op_ptr -> opcode))) ||
                        (comp_ins -> is_long_format == TRUE)) {
                /* if branch target, then even NOP has to be included. */
                        current_opnum++;
                        size = get_ith_size(comp_ins, i);
                        if (size > 26){
                                extension_start[i] = extension_position;
                                extension_position += (size - 26);
                                total_num_bits_in_ins += (size - 26);
                        }
                        twenty_four_bit_start[i] = position;
                        two_bit_start[i] = current_2bit_position;
                        current_2bit_position -= 2;
```

```
                        position +=24;
                        if (((current_opnum - num_ops_in_first_group) % 4) == 0)

                        {
                                position += 8;
                                current_2bit_position = position - 2;
                        }
                }
        }


        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                op_ptr = &(comp_ins -> operation[i]);
                if ((!(IS_NOP(op_ptr -> opcode))) ||
                        (comp_ins -> is_long_format == TRUE)){
                size = get_ith_size(comp_ins, i);
                bits_translate(tmp_2, i*BYTES_IN_ONE_COMP_OP*8, 24,
                        comp_ins -> final_instruction, twenty_four_bit_s
tart[i]);

                bits_translate(tmp_2, i*BYTES_IN_ONE_COMP_OP*8+24, 2,
                        comp_ins -> final_instruction, two_bit_start[i])
;

                if (size > 26){
                        bits_translate(tmp_2, i*BYTES_IN_ONE_COMP_OP*8+
26,

                                (size - 26), comp_ins -> final_instructi

on,

                                extension_start[i]);
                }
                }
        }
    /* Now the final instruction is having the bits in the
            litle endian ordering with the lsb of each byte occurring
            at the left (which is the msb of the sparc). As per the
            disucussion with Eino on Aug 16th, noted in my notebook
            and to be written up somewhere soon, we need to reverse
            the bits in each byte, so that the lsb of each byte matches
            with the lsb of the machine.

     Also, the address field formats need to be updated.
    */
            comp_ins -> num_bytes = total_num_bits_in_ins/8;
            if (total_num_bits_in_ins % 8 != 0) comp_ins -> num_bytes++;

            for (i=0; i < comp_ins -> num_bytes; i++){
                comp_ins -> final_instruction[i] =
                        reverse_bits(comp_ins -> final_instruction[i]);
            };

            for (slot=0; slot < NUM_ISSUE_SLOTS; slot++){
                op_ptr = &(comp_ins -> operation[slot]);
                if (op_ptr -> address_field_format != NO_ADDRESS) {
                        scatter_descr_cpy(&local_descr,
                                &FORMAT_TABLE.scatter_table[op_ptr -> address_fi
eld_format]);

                                for (j=0; j < local_descr.num_entries; j++){
                                /* We assume that the 24 bit offset exists in
                                        the format and that it is of width 2.
                                        We will further assume that src_offsets

of

                                        26 and 34 exist for now and are of lengt

hs 5

                                        and 8*/
                                        if (local_descr.table[j].src_offset == 2

4){

                                                local_descr.table[j].src_offset
=

                                                        two_bit_start[slot];
                                        }else if (local_descr.table[j].src_offse

t == 26){

                                                local_descr.table[j].src_offset

=

                                                        extension_start[slot];
                                        }else if (local_descr.table[j].src_offse

t == 34){

                                                local_descr.table[j].src_offset

=

                                                        extension_start[slot]+8;

                                        }else{
                                        local_descr.table[j].src_offset +=
                                                twenty_four_bit_start[slot];
                                        }
                                }
                        reverse_scatter_descr(&local_descr, &reverse_descr);
                        /* Now everything for the address field format is
                        in place(i.e., refers to the position in the final
                        instruction) except that the bits get reversed, which
                        needs to be taken into account. Once that is done,
                        we can enter this in the scatter table. */
                                for (pos=0; pos < reverse_descr.total_width; pos

++){

                                        if ((src_pos = get_position_in_src(pos,
                                                &reverse_descr)) < 0){
                                                LOG_ERROR("Illegal value returne

d by get_position_in_src()");

                                        }
                                        new_descr.table[pos].dst_offset = pos;
                                        new_descr.table[pos].width = 1;
                                        new_offset = (src_pos - (src_pos % 8)) +

                                                (7 - (src_pos % 8));
                                        new_descr.table[pos].src_offset =  new_o

ffset;

                                }
                                new_descr.num_entries = reverse_descr total_widt

h;

                                new_descr.total_width = reverse_descr.total_widt

h;

                                collapse_scatter_descr(&new_descr, &collapsed_de

scr);

                                scat_id = scatter_table_lookup(input_s_table, &c

ollapsed_descr);

                                if (scat_id < 0) {
                                        scat_ptr = get_free_scatter_descr(input_

s_table);

                                        scatter_descr_cpy(scat_ptr, &collapsed_d

escr);

                                        scat_id = scatter_table_lookup(input_s_t

able, &collapsed_descr);

                                }
                                op_ptr -> address_field_format = scat_id;
                        }
                }
        }


/*----------------------dressup_ins_on_the_fly()----------------------------*/
/*
        Given all teh components of the final compressed instruction,
        as obtained by get_compressed_ops(), we still need to combine
        all the information, and put them into comp_ins->final_instruction
```

```
        in the way the instruction gets written into the compressed file.
        This involves combining the format bits and the operation bits,
        and moving bits around in the instruction based on the 24-bit
        part, 2-bit part and the extension part information.
        This also means that the scatter types of the addresses need
        recomputing and appropriate entries made in the output scatter
        table. (which is unfortunately called input_s_table here!)

        NOTE: Here the context_bits are from the next instruction.
        If context_bits is NULL CONST_FORMAT is used.
*/

void
dressup_ins_on_the_fly(CompressedIns *comp_ins, byte *context_bits,
                       ScatterTable *input_s_table)
{
char error_str[MAX_LENGTH_OF_ERROR];
byte tmp_1[MAX_BYTES_IN_COMP_INS];
byte tmp_2[MAX_BYTES_IN_COMP_INS];
int i, j;
OperationDescr *op_ptr;
int size;/*tmp variable for size of an operation. */
ScatterDescr local_descr, *scat_ptr;
int current_position; /*used for copying operrations. */
int scat_id; /*scatter id - used to do scatter table lookup. */
ScatterTriple s_arr[MAX_SCATTER_TABLE_SIZE];
byte nop_bits[] = {0xff};

        local_descr.table = s_arr;
        for (i=0; i < NUM_ISSUE_SLOTS; i++){
                op_ptr = &(comp_ins -> operation[i]);

                        tmp_1[SRC_1_BYTE] = op_ptr -> arg1;
                        tmp_1[SRC_2_BYTE] = op_ptr -> arg2;
                        tmp_1[DST_BYTE]   = op_ptr -> dest;
                        tmp_1[GUARD_BYTE] = op_ptr -> guard;
                        for (j=0; j < BYTES_PER_MODIFIER; j++){
                                tmp_1[MODIFIER_BYTE + j] = op_ptr -> modifier[j];
                        };
                        for (j=0; j < BYTES_PER_OPCODE; j++){
                                tmp_1[OPCODE_BYTE + j] = op_ptr -> comp_opcode[j];
                        };
                        local_extract_and_arrange_bits(tmp_1, 0,
                                &(FORMAT_TABLE.scatter_table[op_ptr -> format]),
                                tmp_2 + i*BYTES_IN_ONE_COMP_OP,
                                opcode_sign[op_ptr -> opcode],
                                opcode_latency[op_ptr -> opcode]);

/*
                        extract_and_arrange_bits(tmp_1, 0,
                                &(FORMAT_TABLE.scatter_table[op_ptr -> format]),
                                tmp_2 + i*BYTES_IN_ONE_COMP_OP);
*/
                        op_ptr -> address_field_format = get_address_format(op_ptr -> fo
rmat, &FORMAT_TABLE);
                        if (address_opcode[op_ptr -> opcode]) {
                                if (op_ptr -> address_field_format == NO_ADDRESS){
                                        sprintf(error_str,"Address field format not avai
lable for opcode %d.", op_ptr -> opcode);
                                        LOG_ERROR(error_str);
                                }
                        }
                        /* The above stmt gives the basic address format, within
                                an operation. i.e., the format assuming that the
                                bits in the operation are not reordered according to
                                24-bit part, 2-bit part etc. The reordering results
                                in another level of scattering and that will be
                                handled by the reordering code.  The final address
                                format will be w.r.t. the beginning of the instruction.
                                */
                        if(op_ptr -> opcode == IJMPI_OPCODE){
                                fill_const(tmp_2 +i*BYTES_IN_ONE_COMP_OP,31 , 1,CONST_1)
;
                        }

        }

        /* The BNF stuff goes here. This also handles the format bits.
                A NULL context_bits => CONST_FORMAT. */
        /* for now we assume that there is no reordering. The context
                bits are placed first, and then the operations are
                placed one after the other. This would mean that the
                address formats now just get translated by some amount. */
                if (context_bits != NULL){
                        bits_translate(context_bits, 0, 2*NUM_ISSUE_SLOTS,
                                comp_ins -> final_instruction, 0);
                } else {
                        bits_translate(CONST_FORMAT, 0, 2*NUM_ISSUE_SLOTS,
                                comp_ins -> final_instruction, 0);
                };
                current_position = 2*NUM_ISSUE_SLOTS;
                for (i=0; i < NUM_ISSUE_SLOTS; i++){
                        op_ptr = &(comp_ins -> operation[i]);
                        if (!(IS_NOP(op_ptr -> opcode))) {
                                size = get_ith_size(comp_ins, i);
                                bits_translate(tmp_2, i*BYTES_IN_ONE_COMP_OP*8, size,
                                        comp_ins -> final_instruction, current_position)
;
                                if (op_ptr -> address_field_format != NO_ADDRESS) {
                                        if (DO_REVERSE){
                                                reverse_scatter_descr(&FORMAT_TABLE.scat
ter_table[op_ptr -> address_field_format], &local_descr);
                                        }
                                        /* The above reversal is needed because the
                                        addresses are to be interpreted by the linker in

                                        the msb to the right fashion, whereas they are
                                        reperesented here in the msb to the left fashion

                                        */

                                        for (j=0; j < local_descr.num_entries; j++){
                                                local_descr.table[j].src_offset +=
                                                        current_position;
                                        }
                                        scat_id = scatter_table_lookup(input_s_table, &l
ocal_descr);
                                        if (scat_id < 0) {
                                                scat_ptr = get_free_scatter_descr(input_
s_table);
                                                scatter_descr_cpy(scat_ptr, &local_descr
);
                                                scat_id = scatter_table_lookup(input_s_t
able, &local_descr);
                                        }
                                        op_ptr -> address_field_format = scat_id;
                                }
                                current_position += size;
                                } else {
                                        if (comp_ins -> is_long_format == TRUE){
                                                size = get_ith_size(comp_ins, i);
                                                bits_translate(tmp_2, i*BYTES_IN_ONE_COMP_OP*8,
size,
```

```
                                        comp_ins -> final_instruction, current_p
osition);
                                    current_position += size;

                                }
                            }
                        }
                        comp_ins -> num_bytes = current_position/8;
                        if (current_position % 8 != 0) comp_ins -> num_bytes++;
}
```

```
/*****************************************************************/
/*                      File: comp_bitstring.c
                        */
/*                      Author: Hari Hampapuram
                        */
/*****************************************************************/
/* Associated files:
        Routines defined in comp_utils.c, comp_scheme.c, comp_reference.c
        comp_btarget.c are called.
*/
/* Functions defined in this file are:
                1. void
                        compress_bitstring(UncompressedBitstring *uncomp_bitstring,
                                                        CompressedBitstring *com
p_bitstring,
                                                        BtargetTable *bt
arget_table,
                                                        ReferenceTable *
ref_table ,
                                                        ScatterTable *sc
atter_table);
                2. void update_bitfields(byte * bitstring_base,
                                                        ReferenceTable *
ref_table,
                                                        BtargetTable *bt
arget_table,
                                                        Module mod);
                3. get_opnum(unsigned long position,
                                        unsigned long current_ins_num); This sho
uld
                                probably be static.

There is also get_ith_scatter_type() which definitely is static.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/******* files from linker **********/
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "inum_map.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"


/******* files from compressor ********/
#include "compressor.h"
#include "comp_bitstring.h"

#include "comp_btarget.h"
#include "comp_misc.h"
#include "comp_reference.h"
#include "comp_scatter.h"
#include "comp_scheme.h"
#include "comp_utils.h"
#include "dump_structs.h"

/*------------------get_opnum()------------------------------*/
/* position  -        is the bit offset in the bitstring and
   current_ins_num -  is the instruction number (i.e. the address
in the address space for text section.)
        The function returns the issue slot in which the position
        occurs and OUT_OF_INSTRUCTION otherwise. */

get_opnum(unsigned long position, unsigned long current_ins_num)
{

unsigned long  base_bit_num,offset,num;

        base_bit_num = current_ins_num *(INSTRUCTION_WIDTH_BITS);
        if ((offset = position - base_bit_num) < 0){
                return(OUT_OF_INSTRUCTION);
        };

        if ((num = offset / OPERATION_WIDTH) < NUM_ISSUE_SLOTS){
                return (num);
        }
        return(OUT_OF_INSTRUCTION);
}

/*----------------get_ith_scatter_type()--------------------*/
/* comp_ins   - pointer to a compressed instruction. To be
                                meaningful, the struct pointed to by comp_ins
                                must have relevant values (say filled by a
                                call to get_compressed_ops()).
        opnum      - referes to the issue slot or the operation number
                                within the instruction.
        returns the scatter_id of the address field format. The scatter
        id is w.r.t the scatter table which was passed to the functions
        get_compressed_ops and dressup_ins. It is the scatter table of
        the output module.
*/

static
get_ith_scatter_type(CompressedIns *comp_ins, int opnum)
{
        return(comp_ins -> operation[opnum].address_field_format);
}

/*****************************************************************/
/*************          compression                  *************/
/*****************************************************************/

/*----------------append_to_compressed()---------------------------*/
/* comp_bitstring - This is the bitstring to which we add the compressed
                                instructions one by one. The bitstring will get
                                modified as one more instruction is appended.
        next_location - is the byte address from where to start appending
                                the next instruction. Must be a valid address
                                within the space allocated for the compressed
                                bitsitring.
        comp_ins       - must have the correct final instruction. Also,
                                comp_ins->num_bytes must have the correct value.


*/
```

```
void
append_to_compressed(CompressedBitstring *comp_bitstring,
                        byte *next_location, CompressedIns *comp_ins)
{
int i;

        for (i=0; i < comp_ins -> num_bytes; i++){
                *(next_location + i) =
                        comp_ins -> final_instruction[i];
        }
        comp_bitstring -> first_unused_address = next_location +
                        comp_ins -> num_bytes;
}

/*************************************************************/
/*****************compress_bitstring()-***********************/
/*************************************************************/
/* compress_bitstring requires that  reference table be
        initialized for iteration before it is called for the
        first time. (as of now). This can be changed so that
        initially, within in this function we iterate through the
        reference table until we reach the instruction that is
        first in the uncompressed  bitstring. */
/* uncomp_bitstring - begin_address should be byte address of the
                        uncompressed bitstring from where the compression
                        is to start.
                                - num_instructions should be the number of
                        instructions beginning at begin_address that are to be
                        compressed.
                                - first_ins_num should be the address (in addres
s
                        space of the text section i.e the nistruction number)
                        of the first instruction in the uncompressed bitstring.
                                - current_address should be the byte address
                        of the first instruction, ie, should be begin_address.
        comp_bitstring - begin_address should be the address of the
                        frist byte in the compressed code's text section.
                                - first_unused_address should be the address of
the
                        byte from where one can possibly start appending code.
                                - begin_last_src_file and size_last_src_file wil
l be
                        set by this function to be byte * and the size in num of
                        bytes of the part of the code corresponding to the current
                        source file.
        btarget_table - must be initialized to the table of branch targets.
        ref_table     - must be the reference table of the text section.
        scatter_table - this is the scatter_table into which any
                        scatter_types that become part of the output will be entered
                        i.e. this must be the scatter_table of the output module.

HOW:1. Get the basic data from uncomp_bitstring;
        2. Compress the first instruction by calling get_compressed_ops().
        3. Step through the remaining instructions by having two pointers,
                prev_ptr and current_ptr. prev_ptr has the compressed instr.
                of the previous instruction, and current_ptr has the format
                bits of the current instruction. These are combined to get the
                final compressed instruction.
        4. The last instruction will have no "next instruction" and hence
                we use constanr format bits for this.
        5. Each time an instruction is compressed, the reference table
                entries (if any) taht fall into this instruction are updated.
                They will be needed for updating the bitfields and also for
                outputting the final module.
        6. Each time an instruction is compressed, the btarget_table is
                updated if the current instruction is a branch target. The
                new byte offset is entered as a new address corresponding to
                the old instruction number.
*/

void
compress_bitstring(UncompressedBitstring *uncomp_bitstring,
                CompressedBitstring *comp_bitstring,
                        BtargetTable *btarget_table,
                        ReferenceTable *ref_table ,
                        ScatterTable *scatter_table,
                   InumMap *inum_map)

{
CompressedIns comp_ins_1, comp_ins_2;
CompressedIns *current_ptr, *prev_ptr, *tmp_ptr;
byte *  next_location;
RefDescr *current_reference;
BitAddr new_offset, old_offset;
Address ins_address, ins_address_2;
unsigned long current_ins_num, operation_num;
unsigned long uncomp_offset, tmp_value;
unsigned long num_instructions, idx;


        num_instructions = uncomp_bitstring -> num_instructions;


        if (num_instructions == 0) {
                comp_bitstring -> begin_last_src_file =
                        comp_bitstring -> first_unused_address;
                comp_bitstring -> size_last_src_file = 0;
                return;
        }
        current_reference = get_current_reference(ref_table);
        current_ins_num = uncomp_bitstring -> first_ins_num;
        bit_vector_to_Address(&ins_address, sizeof(unsigned long)*8, (byte *)&current_in
s_num);
        prev_ptr = &comp_ins_1;
        current_ptr = &comp_ins_2;
        prev_ptr -> is_btarget =
                (btarget_table_lookup(btarget_table, &ins_address) != NULL);
                /* In fact a serious error - How can the first
                instruction of a file be not a branch target? Maybe
                for a start address, but that I guess should be considered
                a branch target too.
                */
        get_compressed_ops(uncomp_bitstring -> current_address, &comp_ins_1,
                                prev_ptr -> is_btarget);
        comp_bitstring -> begin_last_src_file =
                get_address_for_next(comp_bitstring, prev_ptr -> is_btarget);

#if DEBUG
        printf("-----------------------------------------------\n");
                {int tst, tst1;
                for (tst1 =0; tst1 < NUM_ISSUE_SLOTS; tst1++){
                for ( tst=0; tst < OPERATION_WIDTH/8 +1; tst++){
                        write_byte(stdout,uncomp_bitstring -> current_address[tst +tst1*
(OPERATION_WIDTH/8 +1)]);
                        printf(" ");
                }
                printf("\n");
                }
                printf("\n");
                }
#endif
```

```
        for (idx=1; idx < num_instructions; idx++)      {
                uncomp_bitstring -> current_address += INSTRUCTION_WIDTH_BYTES;
        /*-------------------------------------------------*/
        /*      Part I. get the compressed operations and the
                format bits for the current instruction.                */
        /*-------------------------------------------------*/
                current_ins_num =
                      ,uncomp_bitstring -> current_address - uncomp_bitstring -> begin_
address;
                current_ins_num = current_ins_num / (INSTRUCTION_WIDTH_BYTES);
                current_ins_num += uncomp_bitstring -> first_ins_num;
                bit_vector_to_Address(&ins_address, sizeof(unsigned long)*8,
                                (byte *)&current_ins_num);
                current_ptr -> is_btarget =
                        (btarget_table_lookup(btarget_table, &ins_address) != NULL);
                get_compressed_ops(uncomp_bitstring -> current_address,
                                                current_ptr, current_ptr -> is_btarget);

                dressup_ins(prev_ptr, current_ptr -> context_bits, scatter_table);
#if DEBUG
                {int tst;
                for ( tst=0; tst < prev_ptr -> num_bytes; tst++){
                        write_byte(stdout,prev_ptr -> final_instruction[tst]);
                        printf(" ");
                }
                printf("\n");
                }
#endif
                next_location =
                        get_address_for_next(comp_bitstring, prev_ptr -> is_btarget);
                append_to_compressed(comp_bitstring, next_location,
                                prev_ptr);
                if (G_all_globals.flags.mapon){
                        if (prev_ptr -> is_btarget){
                                fprintf(G_all_globals.mapfile,"***");
                        }
                        fprintf(G_all_globals.mapfile,"Instruction Number = %d, Byte Add
ress = %d %#x\n",
                                current_ins_num - 1, next_location - comp_bitstring->beg
in_address, next_location - comp_bitstring->begin_address);
                }
                inum_map_set_offset(inum_map, (current_ins_num - 1), (next_location - co
mp_bitstring->begin_address));
                inum_map_set_size(inum_map, (current_ins_num -1), (prev_ptr -> num_bytes
));
        /*-------------------------------------------------*/
        /*      Part II. update the reference table.                    */
        /*-------------------------------------------------*/
                while (current_reference != NULL) {
                        operation_num = get_opnum(current_reference -> position , curren
t_ins_num -1);
                        if (operation_num == OUT_OF_INSTRUCTION){
                                break;
                        }
                                current_reference -> scatter_type =
                                        get_ith_scatter_type(prev_ptr, operation_num);
                        if (current_reference -> scatter_type < 0){
                                /*dump_RefDescr(current_reference);*/
                                STD_ERROR("Negative scatter type encountered.");
                        }
                        current_reference -> position =
                                (next_location - comp_bitstring -> begin_address)*8;
                        /* get_ith_scatter_type() gets the index into the scatter_table

                                that will be part of the final object module. */
                                reference_table_next(ref_table, FALSE);
                                current_reference = get_current_reference(ref_table);
                }
        /*-------------------------------------------------*/
        /*  Part III. update the btarget table.                 */
        /*-------------------------------------------------*/
                tmp_value = (next_location - comp_bitstring -> begin_address);
                bit_vector_to_Address(&ins_address, sizeof(unsigned long)*8, (byte *)&tm
p_value);
                uncomp_offset = current_ins_num -1;
                /* current_address has already moved. We need the address
                        of the previous instruction, as it is the previous
                        instruction that is appended at next_location. */
                bit_vector_to_Address(&ins_address_2, sizeof(unsigned long)*8, (byte *)&
uncomp_offset);

                btarget_table_update(btarget_table,
                        &ins_address_2, &ins_address);
        /*-------------------------------------------------*/
        /*  Part IV. swap the pointers for next iteration.     */
        /*-------------------------------------------------*/
                tmp_ptr = prev_ptr;
                prev_ptr = current_ptr;
                current_ptr = tmp_ptr;
#if DEBUG
        printf("-------------------------------------------------\n");
        printf("uncomp_bitstring -> current_address=%d\n", uncomp_bitstring -> current_a
ddress);
                {int tst, tst1;
                for (tst1 =0; tst1 < NUM_ISSUE_SLOTS; tst1++){
                for ( tst=0; tst < OPERATION_WIDTH/8 +1; tst++){
                        write_byte(stdout,uncomp_bitstring -> current_address[tst +tst1*
(OPERATION_WIDTH/8 +1)]);
                        printf(" ");
                }
                printf("\n");
                }
                printf("\n");
                }
#endif

        }
        /*-------------------------------------------------*/
        /*  The last instruction is left out -process that;
                prev_ptr has the instruction (we swapped already).      */
        /*-------------------------------------------------*/
        dressup_ins(prev_ptr, NULL, scatter_table);
        next_location =
                get_address_for_next(comp_bitstring, prev_ptr -> is_btarget);
        append_to_compressed(comp_bitstring, next_location, prev_ptr);
        if (G_all_globals.flags.mapon){
                if (prev_ptr -> is_btarget){
                        fprintf(G_all_globals.mapfile,"***");
                }
                fprintf(G_all_globals.mapfile,"Instruction Number = %d, Byte Address = %
d %#x\n",
                                                current_ins_num , next_location - comp_bitstring
->begin_address, next_location - comp_bitstring->begin_address);
        }
        inum_map_set_offset(inum_map, current_ins_num , next_location - comp_bitstring->
begin_address);
        inum_map_set_size(inum_map, current_ins_num , prev_ptr -> num_bytes);

        /*-------------------------------------------------*/
        /* update the reference table and btarget table as above. */
```

```
        /*-------------------------------------------------------*/
                while (current_reference != NULL) {
                        operation_num = get_opnum(current_reference -> position , curren
t_ins_num);
                        if (operation_num == OUT_OF_INSTRUCTION){
                                break;
                        }
                        current_reference -> position =
                                (next_location - comp_bitstring -> begin_address)*8;
                /*      if (IS_TEXT_SECTION_ID(current_reference -> section,
                                        ref_table -> defmod)){
                */
                                current_reference -> scatter_type =
                                        get_ith_scatter_type(prev_ptr, operation_num);
                /*      }
                */
                        if (current_reference -> scatter_type < 0){
                                STD_ERROR("Negative scatter type encountered.");
                        }
                        /* get_ith_scatter_type() gets the index into the scatter_table

                                that will be part of the final object module. */

                        reference_table_next(ref_table, FALSE);
                        current_reference = get_current_reference(ref_table);
                }
        /*-------------------------------------------------------*/
        /*              update the btarget table.                */
        /*-------------------------------------------------------*/
                tmp_value = (next_location - comp_bitstring -> begin_address);
                bit_vector_to_Address(&ins_address, sizeof(unsigned long)*8,
                                        (byte *)&tmp_value);
                bit_vector_to_Address(&ins_address_2, sizeof(unsigned long)*8,
                                        (byte *)&current_ins_num);

                btarget_table_update(btarget_table,
                        &ins_address_2, &ins_address);

        comp_bitstring -> size_last_src_file =
                comp_bitstring -> first_unused_address -
                comp_bitstring -> begin_last_src_file;
}


/*****************************************************************/
/*************   update bitfields                                ***************
*/
/*****************************************************************/
/*
        1. bitstring_base  - is the byte ptr to teh base of the bitstring
                                        which is being updated. The positions in
 the
                                        refeerence table entries are interpreted

                                        w.r.t this base.
        2. ref_table       - The entries in this table are used to
                                         locate the  fields which get updated. T
he
                                        references in this table that are define
d
                                        in teh text section are updated to their

                                        new value as given by the btarget_table.

        3. btarget_table   - This table is used to obtain the new values
                                        for the branch targets given their old v
alues.
                                        Thus every value referred by a an entry
in the
                                        ref_table must be available in this tabl
e and
                                        the corresponding new address must have
been
                                        initialized.
        4. mod             - The mod is necessary here so that the reference
                                        table entries can be interpreted properl
y
                                        according to the scatter_table of the mo
dule.

The bitstring pointed to by the bitstring_base will be changed.
This function is used to update the bitstrings of the various
sections after compressing the bitstring of the binary section.
*/

void
update_bitfields(byte * bitstring_base, ReferenceTable *ref_table,
                          BtargetTable *btarget_table, Module mod)
{
RefDescr * current_ref;
BtargetDescr    * btarget;
Address address;
byte buffer[MAX_BYTES_PER_ADDRESS + 1];
ScatterDescr *scatter_ptr;
byte * src_base;
int offset;

        reference_table_next(ref_table, TRUE);
        while ((current_ref = reference_table_next(ref_table, FALSE))
                        != NULL)
        {
                if (IS_TEXT_SECTION_ID(current_ref -> section, ref_table -> defmod)){
                        src_base = bitstring_base + (current_ref -> position /8);
                        offset = current_ref -> position % 8;
                        scatter_ptr =
                                scatter_descr_from_id(current_ref -> scatter_type,
                                                                mod -> scatter_t
able);
                        if ((scatter_ptr -> num_entries < 0)||
                                (scatter_ptr -> total_width == 0)){
                                STD_ERROR("Inconsistent scatter descriptor.");
                        }
                        extract_and_arrange_bits(src_base,offset, scatter_ptr, buffer);

                        bit_vector_to_Address(&address, scatter_ptr -> total_width,
                                                                buffer);
                        if ((btarget = btarget_table_lookup(btarget_table, &address))
                                == NULL)
                        {
                                STD_ERROR("Branch table lookup failure.");
                        };
                        Address_to_bit_vector(&(btarget -> new_address), scatter_ptr ->
total_width,
                                                                buffer);
                        rearrange_and_insert(src_base, scatter_ptr, buffer, offset);
                }
        }
}
```